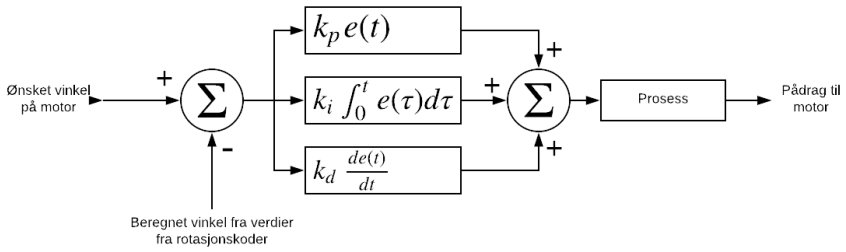
Programmering av servomotor ved hjelp av PID-regulator

Avd. Ing. Espen Teigen

24.08.2018



Figur 1: Blokkskjema for PID-regulator som skal implementeres. e=ønsket verdi - faktisk verdi. E. Teigen, 2018

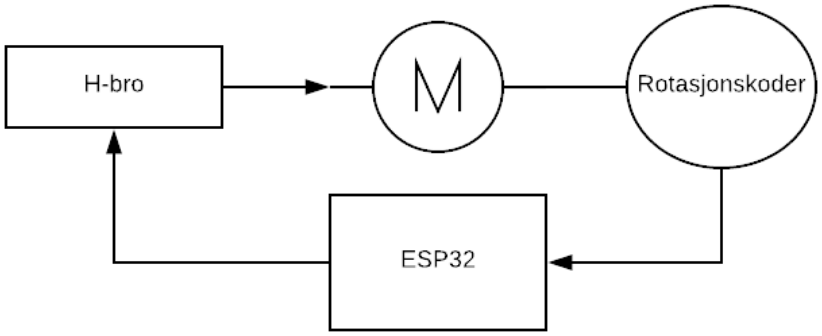
Sammendrag

I denne oppgaven er målet å gi dere en bedre forståelse for hvordan PID-regulering implementeres i praksis. Systemet dere skal jobbe med er laget av deler som er funnet i diverse kroker her på avdelingen, 3D-printet deler som er tegnet i Autodesk Inventor, en H-bro av typen LMD18200, en motor, og en inkrementell rotasjonskoder.

Oppgaven består av to deler. Den første delen blir å kommentere koden som er på mikrokontrolleren, samt skrive PID-reguleringsalgoritmen.

I den andre delen skal dere justere inn K, P og I parametrene og punkt-prøvingsfrekvensen ved hjelp av det medfølgende Windows-programmet.

1 Systemoversikt



Figur 2: Forenklet oversikt over de involverte komponentene i systemet. E. Teigen

Systemet er veldig enkelt, med relativt få komponenter. Men som vi skal se senere, så går det en del energi inn i å utvikle fastvaren.

2 Introduksjon

En servomotor består vanligvis av en DC-motor med en negativ tilbakekobling som brukes til å kontrollere pådrag og rotasjonsretning på motor, noe som gir en veldig bra posisjonskontroll av motoren. De enkleste typene har en elektrisk krets som står for tilbakekobling, mens de mer avanserte kan bruke rotasjonskoder, måling av strøm til motor og PID-regulator. I vårt systemet bruker vi kun rotasjonskoderen som en tilbakemelding om posisjon.

Mesteparten av fastvaren har blitt skrevet på forhånd, slik at dere skal slippe det. Men det kommer likevel til å bli gitt en grundig gjennomgang av hele systemet.

Vi kommer først til å gå gjennom de mekaniske komponentene, deretter de elektriske, så tar vi en kort gjennomgang av koden på esp32.

3 Det mekaniske

3.1 Motoren

Motoren er en DC-motor på ca 12 watt og 12 volt. Det er en giret motor, noe som gjør at den drivende akselen har høyere rotasjonsmoment, men på bekostning av redusert turtall. I denne oppgaven er ikke en høy vinkelhastighet så viktig, og er derfor ikke noe problem.

Så hvordan er fysikken bak hastighetsstyring av en motor? Med dagens transistorteknologi er dette trivielt. Vi bruker puls-bredde modulering(PBM)(pulse-width modulation(PWM) på englsk), på spenningen for å regulere hvor mye strøm som går gjennom spolen. Det er et lineært forhold mellom spenning og turtall, og et lineært forhold mellom moment og strøm på en elektrisk motor gitt at lasten er konstant.

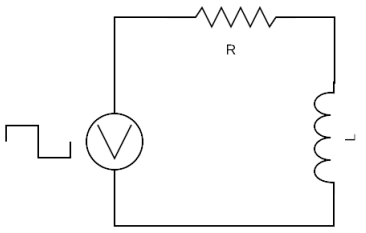
$$\omega = k_v V$$

$$\tau = k_i I$$

Der:
k_v og $k_i$er konstanter spesifikk for motoren
I er strømmen
τ er momentet
ω er vinkelhastigheten

PBM er en firkantbølge som svinger mellom 0 og V_{max} med en konstant frekvens, mens tiden spenningen er på varier innenfor en periode.

Kretsskjemaet for motoren vil se noe slikt ut:



Figur 3: Kretstegning for motor, en typisk RL-krets. E. Teigen

Vi ser litt nærmere på hvordan hastighetsstyring av motoren fungerer. Siden en spole vil motsette seg endringer i strømmen, kan vi anta at strømmen gjennom

spolen er konstant hvis vi har en konstant gjennomsnittspenning og frekvens er høy nok. Gjennomsnittspenningen til et PBM-signal finner vi enkelt. Vi vet at vi kun kan kontrollere spenningen ved å slå den av og på. Hvis vi ønsker å finne denne spenningen avhengig av tiden den er av og på, kan vi relatere dette til forholdstall som vi kaller D. $D = 0$, så er spenningen av hele tiden av en hel periode, $D = 0.5$ så er spenningen på halve tiden av en hel periode og $D = 1$ så spenningen på hele tiden. Gjennomsnittspenningen er derfor:

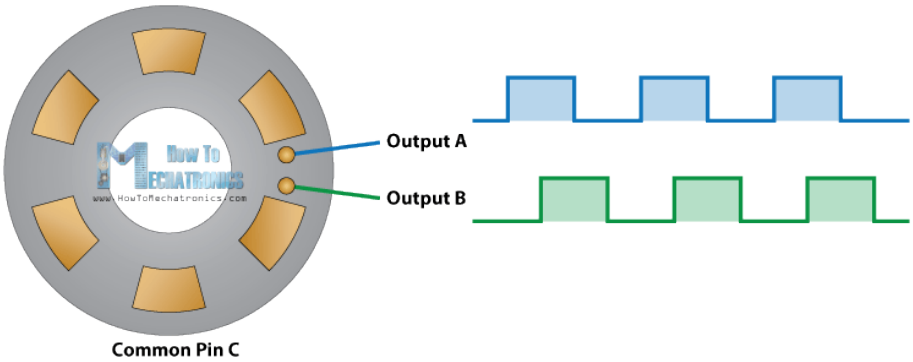
$$V_{gjennomsnitt} = V_{max} D$$

Deretter slenger jeg opp en formel for gjennomsnittstrømmen, som er hentet fra boken Power Electronics av Mohan, Undeland og Robbins(Merk: Denne formelen gjelder kun hvis spenningen går mellom 0 og V_{max}).

$$I_{gjennomsnitt} = \frac{D}{2Lf} V_{max} = \frac{1}{2Lf} V_{gjennomsnitt}$$

Der:
L er induktansen i motoren
f er frekvensen til PBM-signalet

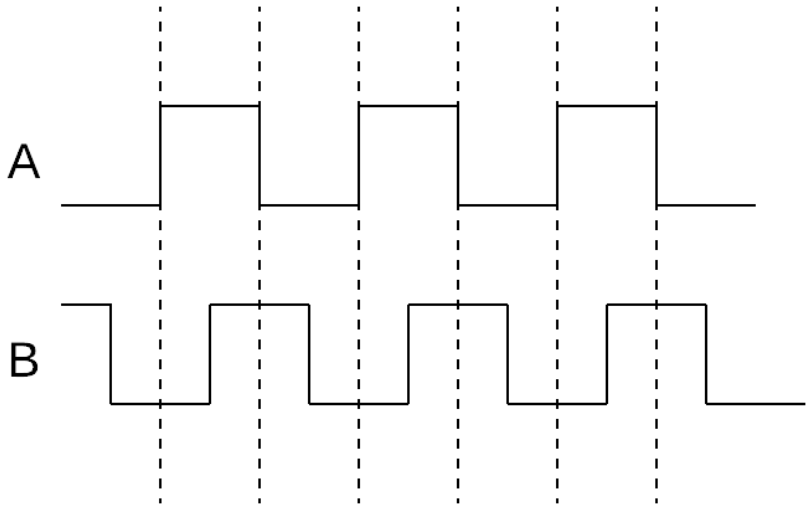
3.2 Inkrementell rotasjonkoder



Figur 4: Konseptskisse for inkrementell rotasjonskoder. Kilde: https://howtomechatronics.com/tutorials/arduino/rotary-encoder-works-use-arduino/

Akselen på en inkrementell rotasjonskoder kobles til den roterende akselen på motoren, og roterer derfor med motoren i samme vinkelhastighet som motoren. Inkrementelle rotasjonskodere, gir ikke informasjon om posisjonen til akselen, men endring i posisjon, derav navnet inkrementell.

Rotasjonskoderen har tre utganger som vanligvis navngis med A, B og P, men dette kan variere fra produsent til produsent. A og B er spenningen fra to pulstog fra rotasjonskoderen, som har en faseforskyvning på 180° relativt til hverandre(se figur 5). P er en puls som gir informasjon om når det har gått en rotasjon, denne trenger ikke vi å bruke.



Figur 5: Pulstog fra rotasjonskoder

Ved å tolke pulstogene fra A og B riktig, kan vi finne både rotasjonsretning, absolutt posisjon og vinkelhastighet.

3.2.1 Hvordan finne rotasjonsretningen

Vi starter med å tolke A og B som to bit, og skriver opp alle mulig kombinasjoner sekvensielt. Vi må se for oss at vi har rotasjon med klokken og mot klokken, dette tilsvarer da og bevege seg mot høyre eller venstre i figur 5. Hvilken vei som er hva, er ikke så interessant akkurat nå.

Med klokken		Mot klokken	
A	B	A	B
0	0	0	0
1	0	0	1
1	1	1	1
0	1	1	0

Vi gjenkjenner dette som gray-kode, der det er kun ett bit som bytter verdi ved hver inkrementering. En annen fordel med gray-kode, er at vi kan også se om den teller den ene eller andre veien, siden rekkefølgen på bitstrømmen fra rotasjonskoderen er unik for rotasjonsretningen hvis vi ser på nåværende og foregående verdier. Kombinasjonen 00 og med etterfølgende 10, så vet vi at rotasjonen er med klokken, siden dette er en unik kombinasjon for rotasjonsretningen.

3.2.2 Hvordan finne absolutt posisjon

Å bruke uttrykket absolutt posisjon, er en sannhet med modifikasjoner, vi kan kun vite absolutt posisjon fra etter vi har slått på systemet og begynte å ta hensyn til endringene som blir gjort. Det er derfor vanlig at ved bruk av inkrementell rotasjonskoder til posisjonsmåling, at vi har en mekanisk bryter eller lignende på ett sted som gir oss en indikasjon på en gitt posisjon, Vi kan også bruke P-signalet for å kalibrere systemet. I vårt tilfelle, er ikke dette inkludert, da vi også kan visuelt gi en kalibrering ved å sette pilen i ønsket posisjon, eller ved hjelp i fra potmeteret og huke av for potmeter i windowsprogrammet.

I vårt tilfelle, så har rotasjonskoderen en oppløsning på 0.6°. Dette vil bety at for hver puls vi får fra A eller B, så har akselen beveget seg $\pm 0.6°$, med litt ekstra usikkerhet. Dette har selvfølgelig en ulempe, og det er at vi ikke vet hvor akselen er mellom disse posisjonene. Rotasjonskoderen kan vi derfor kalle en diskret posisjonsmåler.

3.2.3 Hvordan finne vinkelhastigheten

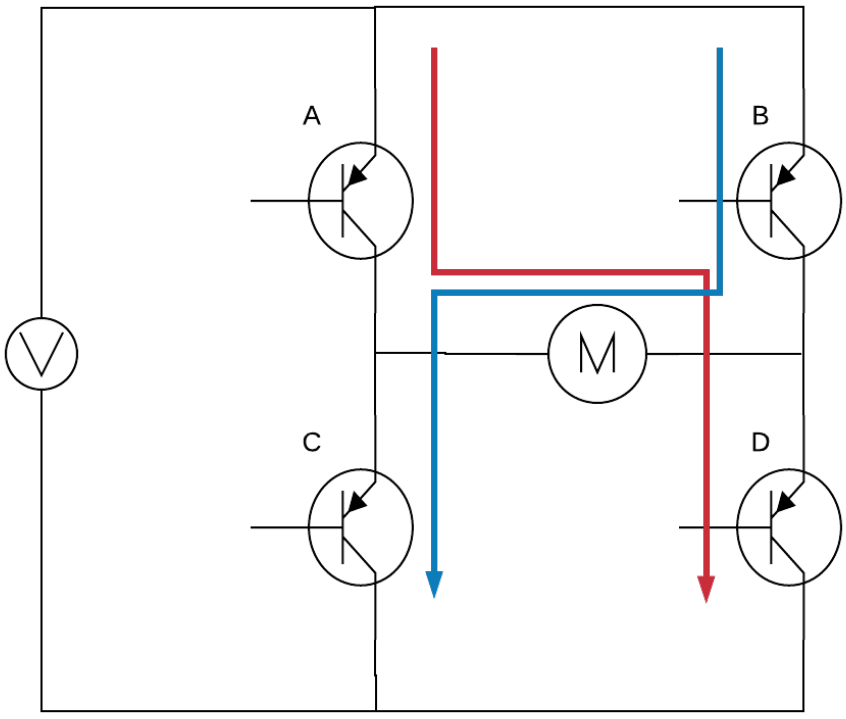
Vinkelhastigheten er ikke relevant for oppgaven, men er relativt enkelt å finne hvis dere er interessert i det. Det er to måter som gjelder. Måle tiden mellom to eller flere pulser, fungerer bra på systemer med lav vinkelhastighet eller måle antall pulser i et kjent tidsintervall, fungerer bra ved høy vinkelhastighet. Hva som er lav og høy hastighet, defineres ut fra hvor høy klokkehastigheten på mikrokontrolleren er, antall pulser pr. rotasjon på koderen og maks hastighet.

4 Det elektriske

4.1 H-bro

Det elektriske er relativt enkelt. Det blir brukt en ferdig H-bro av typen LMD18200. En H-bro bruker 4 transistorer for å styre retningen til strømmen gjennom motoren.
Vi kan også bruke PBM for å regulere hastigheten. Ta gjerne en titt i databladet til LMD18200. Denne blir forsynt med 12 volt fra en gammel strømforsyning fra en PC. Kontrollsignalet som brukes til å bestemme strømretning og PBM kommer fra ESP32-mikrokontrolleren.

Så hvordan kan en H-bro gjøre at en motor kan bytte retning? I sin enkleste form, benyttes fire transistorer. Se figur 6



Figur 6: H-bro i sin enkleste form. Denne kretsen kan ikke brukes på grunn av induktansen til motoren, men gir en god visuell forståelse. E. Teigen 2018

Den røde pilen representerer strømretningen når transistor A og D er ledende, den blå pilen er når transistor B og C er ledende. Ved å styre transistorene, kan man også styre strømretningen igjennom spolen på DC-motoren. Transistorene

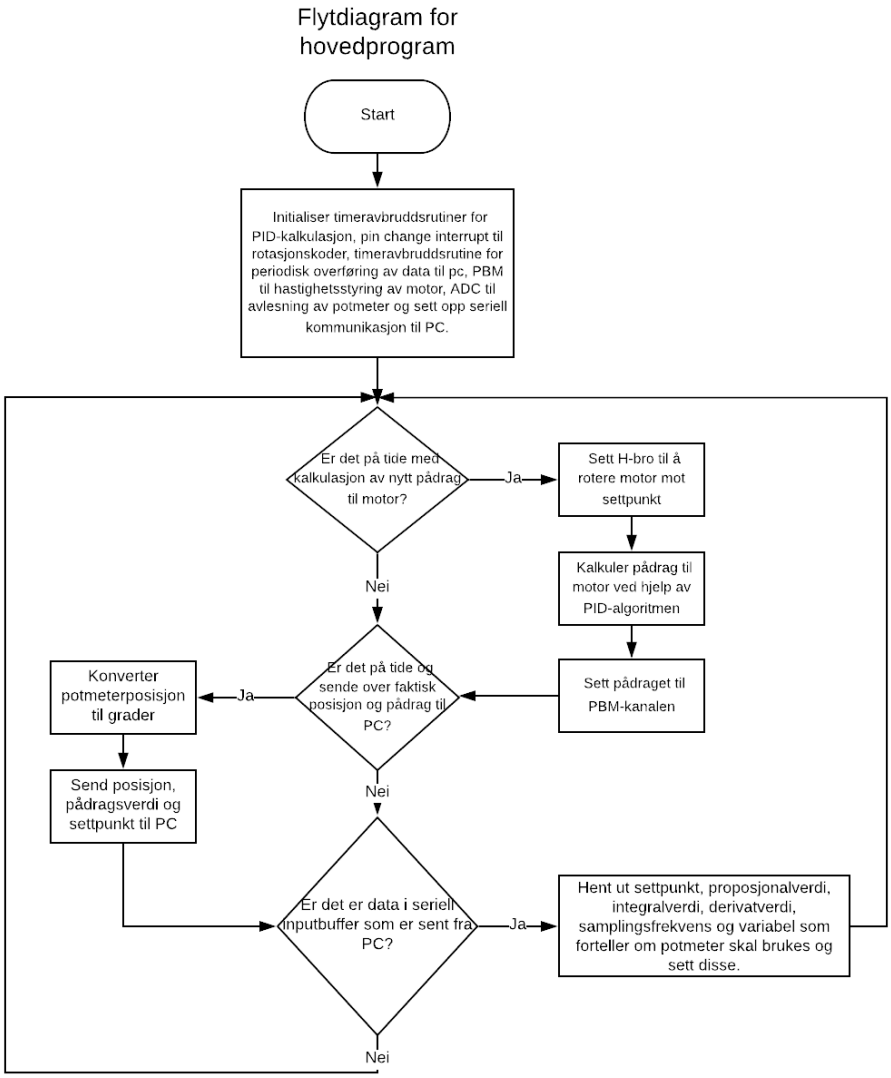
i H-broen, kan være i andre typer konfigurasjoner som gjør at vi kan bremse motoren eller la den snurre fritt. Hvis dere ønsker og vite mer om dette, kan dere titte i databladet til LM18200. Dette er egentlig alt det elektriske som tar seg av styringen av motoren.

5 Oppsett av software

I mappen PID-lab ligger det tre mapper. PID_program er microcontroller-koden, PID er biblioteket der dere skal skrive deres egen PID-regulator For å bruke PID-biblioteket, så må denne mappen legges i library-mappen til arduino. Motor_pid_GUI er et grafisk brukergrensesnitt som dere skal bruke til å tune regulatoren. Dette er skrevet i visual studio, men dere trenger ikke å ha det installert, bare gå til Motor_pid_GUI ⇒ motor_pid_GUI_V2 → Motor_pid_GUI_V2.exe

6 Kode på esp32

6.1 Hovedprogrammet



Figur 7: Hovedflytskjema for programflyten. Merk:Meget forenklet. E. Teigen 2018

6.2 Forklaring av funksjonene som blir brukt

6.2.1 Avbruddsrutiner

IRAM_ATTR er det som kalles en avbruddssubrutine i esp32-verdenen, og er tilsvarende ISR i Atmel/arduino-verdenen. Dette vil si at når en gitt handling

oppstår, så lagres adressen hvor vi er i programmet, og funksjonen som er knyttet opp til handlingen blir kjørt, uavhengig av hva som skjer i koden. Pekeren som forteller hvor vi er i hovedprogrammet blir lagret automatisk, sub-rutinen blir kjørt, og programmet fortsetter der det slapp. En avbruddssubrutine skal være raskest mulig, siden den hindrer resten av programmet å kjøre.

Formatet på teksten videre i dokumentet er nå slik. Først så kommer koden til funksjonen i sin helhet, deretter kommer det en forklaring av koden.

```
//denne funksjonen finner dere i file encoder.ino
void encoderInit () {
  //init pin change interrupt on encoder pin A and B
  pinMode( ENCODER_A, INPUT_PULLUP );
  attachInterrupt( digitalPinToInterrupt( ENCODER_A ), encoderPulse
    , CHANGE );
  pinMode( ENCODER_B, INPUT_PULLUP);
  attachInterrupt( digitalPinToInterrupt( ENCODER_B ), encoderPulse
    , CHANGE );
}
```

I denne funksjonen blir det satt opp pin-change interrupt på pinnene som A og B fra mikrokontrolleren er koblet til. Det blir slått på intern pull-up motstand i mikrokontrolleren for å sørge for at vi ikke har flytende spenninger. encoderPulse er en event-handler som er satt opp med linjen:

```
//Dette ligger i hovedkoden siden den er gloabl
portMUX_TYPE encoderInitMux = portMUX_INITIALIZER_UNLOCKED;
```

For å gjøre den global, er den satt opp før void setup(). CHANGE betyr bare at vi trigger avbruddsrutinen på både fallende og stigende flanke av spenningen. Vi trenger encoderInitMux for å kunne slå av og på avbruddsrutinen

Legg merke til at samme sub-rutine(encoderPulse) blir koblet til begge pinnene. Dette gjør at den samme sub-rutinen blir kjørt både hvis A eller B har en endring.

A og B er definert slik i programmet

```
#define ENCODER_A 32      //Input Pin A on encoder
#define ENCODER_B 15      //Input pin B on encoder
```

Hvor tallene er inngangene som blir brukt på mikrokontrolleren.

```
//Denne funksjonen finner dere i filen encoder.ino
void IRAM_ATTR encoderPulse () {
  //turn of interrupt, so we can finish the calculation before we
    //leave the interrupt routine
  portENTER_CRITICAL_ISR ( &encoderInitMux );

  encoderState =
  (
  0x0F
  & ( encoderState << 2 ) )
  | ( ( 1 << digitalRead( ENCODER_A ) )
  | digitalRead( ENCODER_B )
  );

  if (
   encoderState == 0x02
  || encoderState == 0x04
  || encoderState == 0x0B
  || encoderState == 0x0D ){

    encoderDir = -1;
    encoderPos -= ENCODER_RESOLUTION;
  }

  else if (
     encoderState == 0x01
  || encoderState == 0x07
  || encoderState == 0x08
  || encoderState == 0x0E ) {

    encoderDir = 1;
    encoderPos += ENCODER_RESOLUTION;
  }
  //turn on interrupts
  portEXIT_CRITICAL_ISR ( &encoderInitMux );
}
```

Dette er sub-rutinen som ble satt opp i encoderInit(). Hver gang A eller B fra rotasjonskoderen har en endring i tilstand, blir denne kodesnutten kjørt. Variablene som blir brukt her er globale og er definert slik:

```
//Dette ligger i hovedkoden siden de er gloabel
//Variable that holds the current and previous position
volatile uint8_t encoderState = 0;
//Start position of encoder
volatile double encoderPos = ENCODER_START_POS;
//Direction of rotation of encoder
volatile int8_t encoderDir;
```

"volatileer ett nøkkelord i C-programmeringsspråket som forteller kompilatoren at denne variabelen kan endre seg når som helst. Alle variabler som blir endret i en avbruddssubrutine skal alltid være volatile.

Viktig: Her ser dere at jeg har skrevet en del kode i sub-rutinen. Dette har jeg gjort for at jeg vet at jeg har nok ressurser tilgjengelig til å fullføre koden. Det skal også nevnes at å finne posisjon har høyst prioritet, da alt annet er

avhengig av denne verdien.

```
//Denne funksjonen finner dere i serialCommunication.ino
void SerialTimerInit() {
  SerialTimerHW = timerBegin(1, 80, true);

  timerAttachInterrupt(SerialTimerHW, &SerialTimer, true);
  //timer is set to trigger with 25Hz
  timerAlarmWrite ( SerialTimerHW, 25000  , true );

  timerAlarmEnable ( SerialTimerHW );
}
```

Jeg ønsker å sende målte verdier og kalkulerte verdier over til en PC med en frekvens på 40Hz. Jeg setter derfor opp en timer-avbruddsrutine for å håndtere dette. Det denne gjør, er å telle antall klokkepulser, og trigge avbruddsrutinen når vi når en verdi som vi bestemmer for å få 40Hz

SerialTimerHW er en struct som brukes til konfigurasjon av timer-avbruddsrutinen. timerBegin(1, 80, true), betyr at vi ønsker å bruke timer 1, at vi ønsker å dividere klokkefrekvensen på 80 og true betyr at ønsker at timeren skal telle oppover. Vi sender denne structen sammen med navnet til avbruddssubrutinen Serialtimer(). true betyr at vi ønsker å aktivere avbruddsrutinen.

I timerAlarmWrite(SerialTimerHW, 25000, true), så setter vi hvor mange pulser som skal telles, true gjør at telleren nullstiller seg hver gang en avbruddsrutine blir trigget. Formelen for å finne hvor mange pulser vi skal telle, er relativt enkelt å finne.

esp32-brettet har en klokkehastighet på 80MHz, hvis vi dividerer denne verdien med 80, får vi en teller, som teller opp med en hastighet på 1MHz. Så hvis vi da ønsker å få avbrudd 40 ganger i sekundet, trenger vi bare å finne ut hvor mange pulser dette er. Vi vet at vi har 1 million pulser pr sekund, så hvis vi dividerer antall pulser pr. sekund, med ønsket frekvens, får vi antall pulser vi ønsker å telle opp til for å trigge 40 ganger i sekundet.

$$pulser = \frac{10^6}{40} = 25000$$

Det er noen ting som må gjøres før denne funksjonen kan kjøres. Det er å lage event-handlers som kompilatoren bruker for å sette opp timeren. De er:

```
//Dette finner dere i hovedkoden, siden disse er globale
//timer interrupt serial data out
portMUX_TYPE SerialTimerMux = portMUX_INITIALIZER_UNLOCKED;
hw_timer_t * SerialTimerHW = NULL;
```

De er satt globale, slik at jeg har tilgang til de i hele programmet, da jeg ikke kan sende verdier inn i en avbruddssubrutine.

```
//Denne funksjonen finner dere i filen serialCommunication.ino
void IRAM_ATTR SerialTimer() {
  portENTER_CRITICAL_ISR ( &SerialTimerMux );
  SerialFlagg = true;
  portEXIT_CRITICAL_ISR ( &SerialTimerMux );
}
```

IRAM_ATTR SerialTimer() Er en veldig enkel sub-rutine. Hver gang ett avbrudd blir registrert på timer 1 i esp32, så settes den globale verdien volatile bool SerialFlagg() = true. Dette flagget blir pollet av en if-setning i void loop().

Det har blitt valgt å bruke et flagg, siden overføring av data, ikke er veldig tidskritisk for prosessen.

```
//Denne funksjonen finner dere i filen pidTimer.ino
void pidInit () {
      //init timer interrupt for encoder timing, use timer 0,
    prescaler 80 and true=count up
      PIDTimerHW = timerBegin( 0, 80, true );
      timerAttachInterrupt( PIDTimerHW, &PIDTimer, true );
    }
```

Vi ønsker å kalkulerer ny verdi på PID i en fast periode T, når PID blir kalkulert, er bestemt av event-handlere som blir satt opp her. Sub-rutinen som blir styrt av denne heter PIDTimer, og vi ser her at adressen til structen PIDTimer blir sendt inn i timerAttachInterrupt. Utenom dette, så er dette bare en vanlig timer-avbruddsrutine som forklart under SerialTimerInit().

Men, på PID-en så ønsker vi å kunne endre samplingsfrekvensen, slik at vi kan justere hvor ofte vi kjører en ny PID-kalkulasjon. Denne frekvensen vil endre responsen i systemet.

```
//Denne funksjonen finner dere i filen pidTimer.ino
void IRAM_ATTR PIDTimer () {
  portENTER_CRITICAL_ISR ( &PIDTimerMux );
  PIDFlagg = true;
  portEXIT_CRITICAL_ISR ( &PIDTimerMux );
}
```

Samme som for IRAM_ATTR SerialTimer(). Vi poller PID-flagget i en if-løkke i void loop().

```
//Denne funksjonen finner dere i filen hardwareFunctions.ino
void pwmInit () {
  ledcSetup ( PWM_CHANNEL, PWM_FREQ, PWM_RES );
  ledcAttachPin (PWM_PIN, PWM_CHANNEL );
}
```

For å styre hastigheten på motoren, bruker vi som nevnt tidliger PBM. I dette tilfellet, så bruker jeg ledc i esp32. Dette er en fastvaretimer, som er laget til bruk av dimming av LED-lys. Grunnen til at jeg har valgt å bruke denne funksjonen, er for at den er veldig enkel å sette opp.

Hele oppsette av PBM-en er gjort med disse funksjonene. ledc(), og ledAttachPin(), Verdiene jeg sender inn har jeg satt opp i starten av koden, og er bare noen konstanter

```
//dette finner du i hovedkoden
#define PWM_PIN 13          //Speed control of H-bridge
#define PWM_CHANNEL 0      //Internal reference to PWM-register in
    microcontroller
#define PWM_FREQ 10000       //Frequency of the PWM-signal
#define PWM_RES 10           //Bit-resolution on PWM-signal
#define PWM_BITS 1024
```

Alle disse er selvforklarende, med unntak av PWM_BITS. Denne verdien brukes for at PID-regulatoren ikke skal gi større pådrag enn vi klarer å gi ut.

Jeg har nevnt tidligere, at på et pulsbreddemodulert signal, har vi en firkantbølge med en fast frekvens, som svinger mellom 0 og en eller annen spenning og vi regulerer gjennomsnittspenningen ved å variere hvor lenge spenningen er på i løpet av en periode. I mikrokontrolleren, har vi ikke en kontinuerlig variasjon, vi har diskret steg. I dette tilfellet, har jeg valgt en 10-bits oppløsning, slik at vi kan variere mellom 0 og 1023. Men motoren kan gå begge veier, som reelt sett gir oss en oppløsning på 11 bit.

For å sette verdien på pulsbredden brukes funksjonen:

```
//Bruken av denne finner dere i void loop()
ledcWrite(PWM_CHANNEL, abs(pidValue));
```

Der pidValue er den kalkulerte PID-verdien. ledcWrite tar kun verdier mellom 0 og 1023, og kan derfor ikke ta imot negative verdier, derav bruken av abs(). Rotasjonsretingen til motoren blir satt i funksjonen motorDir(double _angle) som blir forklart senere.

6.2.2 Vanlige funksjoner

```
//Denne funksjonen finner dere i filen LM18200.ino
void motorDir(double _angle) {
  if ( _angle > PIDSetPoint ) {
    digitalWrite(DIR_PIN, HIGH);
  }
  else if ( _angle < PIDSetPoint) {
    digitalWrite(DIR_PIN, LOW);
  }
}
```

Ved hjelp av denne funksjonen, så bestemmer vi når H-broen skal snu motorretningen. Inn i funksjonen motorDir(), sender vi posisjonen som rotasjonskoder, og da selvfølgelig motor er i, og snur strømretningen hvis vi er på feil side av settpunktet til PID-regulatoren. PID-SetPoint er en gloabl variabel, og blir bestemt når strøm bli satt på systemet, eller når potmeter, eller PC endrer den.

Alle konstantene som blir brukt her, er knyttet opp til hvor jeg har tilkoblet H-broen, PWM-pinnen ble satt opp når PWM ble intialisert og blir satt her:

```
//Disse finner dere i hovedkoden
#define DIR_PIN 12
#define BREAK_PIN 27
#define THERMAL_FLAGG_PIN 33
```

THERMAL_FLAGG_PIN er ikke brukt i koden, men er lagt inn som en referanse hvis behovet skulle komme i fremtiden.

```
//Denne finner dere under hardwareFunctions
void adcInit() {
  adc1_config_width(ADC_WIDTH_12Bit);
  adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_11db);
}
```

Her setter vi opp ADC-en som brukes til å lese av verdiene på potmeteret som brukes til flytting av settpunkt. Konstantene som blir brukt her er en del av biblioteket driver/adc.h og er ganske selvforklarende. For å hente verdien som ligger i registert om den analoge spenningen, brukes funksjonen:

```
//Bruken av denne finner dere i void loop()
adc1_get_voltage(ADC1_CHANNEL_0)
```

```
//Denne funksjonen finner dere i filen serialCommunication.ino
int16_t getValueFromSerial() {
  int16_t integerValue = 0;
  int8_t sign = 1;
  char incomingByte = Serial.read();
  if ( incomingByte == 0x2D ) {
    sign = -1;
    incomingByte = Serial.read();
  }
  while (1) {
    if (incomingByte == 0x0A) break;
    integerValue *= 10;
    integerValue = ((incomingByte - 48) + integerValue);
    incomingByte = Serial.read();
  }
  return integerValue *= sign;
}
```

Denne funksjonen brukes til å lese inn data fra PC. Jeg kommer ikke til å forklare denne, da det er en av oppgavene som skal løses.

```
void setup() {
  pidInit();
  setPIDSamplingFrequency(100);
  pid.setLimits(-PWM_BITS, PWM_BITS);
  //Set som random values for PID
  pid.setConstants(10, 10, 10);
  encoderInit();
  H_BridgeInit();
  SerialTimerInit();
  pwmInit();
  adcInit();
  //Start serial communication
  Serial.begin(115200);
}
```

```
//PIDFlagg is toggled as often as the PID frequency ISR is set
  if (PIDFlag == true) {
    //Find out if motor is in a negative or positiv position
    motorDir(encoderPos);
    //Calculate PID value dependent on the setpoint and position
    pidValue = pid.compute(PIDSetPoint, encoderPos);
    //Write PWM value to pwm pin on H-bridge
    ledcWrite(PWM_CHANNEL, abs(pidValue));
    //Reset flagg
    PIDFlag = false;
  }

  //SerialFlagg is toggled as often as the Serial timer ISR is set
  if (SerialFlag == true) {
    //Map and save position of potmeter
```

```
16      posAnalog = map(adc1_get_voltage(ADC1_CHANNEL_0), 0, 4095, 100,
         -100);
17      //Send real time data to PC
18      Serial.print(String("o") +  encoderPos + String("i") + pidValue
         + String("l") + PIDSetPoint + String("\n"));
19      //Reset flag
20      SerialFlag = false;
21
22    }
23    //Smoothing of analogValues from potmeter if use of potmeter is
        selected
24    if ( ( (posAnalog > posAnalogLast + 1) || (posAnalog <
        posAnalogLast - 1) ) && ( potOrPC == true) ) {
25      posAnalogLast = posAnalog;
26      PIDSetPoint = posAnalog;
27    }
28
29    //If there is data in the serial register
30    if (Serial.available() > 0) {
31      char frequency;
32      incomingByte = Serial.read();
33
34      switch (incomingByte) {
35        case 's':
36          PIDSetPoint = getValueFromSerial();
37          break;
38
39        case 'p':
40          pid.setKp((double)getValueFromSerial());
41          break;
42
43        case 'i':
44          pid.setKi((double)getValueFromSerial());
45          break;
46
47        case 'd':
48          pid.setKd((double)getValueFromSerial());
49          break;
50
51        case 'f':
52          setPIDSamplingFrequency((double)getValueFromSerial());
53          break;
54        case 't':
55          if (getValueFromSerial() == 1) potOrPC = true;
56          else potOrPC = false;
57          break;
58      }
59    }
60  }
```

6.2.3 PID-klassen

PID-klassen har det blitt laget ett eget bibliotek for, dette gjør det mulig for dere å bruke PID-regulatoren i andre prosjekter som involverer C++.

For å lage bibliotek i C++, må det minst være to typer filer som ligger i biblioteket .h og .cpp. Jeg går først igjennom .h og deretter .cpp-filen

.h er det som kalles en header-fil. Header-filer brukes som et interface til .cpp-filen som inneholder koden. Vi hopper rett på .h-filen, og forklarer deretter hva som blir gjort, og hvorfor.

```
#ifndef PID_h
#define PID_h

#include "Arduino.h"

class PID {
  private:
    double lastInput;
    double maximum, minimum;
    double sampleTime;
    double kp, ki, kd;
    double output;

  public:
    void setConstants( double _kp, double _ki, double _kd );
    void setKp(double _kp);
    void setKi(double _ki);
    void setKd(double _kd);
    void setLimits( int _minimum, int _maximum );
    void setFrequency(uint16_t frequency );
    double compute(double setPoint, double input);
};
#endif
```

I klassen, ser vi vi at det er definert public og private variabler og funksjoner, syntaksen er annerledes, men utenom det, så er tankegangen helt lik.
Vi starter med de første to linjene:

```
#ifndef PID_h
#define PID_h

//Legg merke til i bunn
#endif
```

Dette gjør at hvis biblioteket blir importert flere ganger, så vet kompilatoren dette, og tar den ikke med på de andre kallene.

```
#include "Arduino.h"
```

I biblioteker, så må vi inkludere alle biblioteker som blir brukt, og siden vi bruker Arduino-syntax noen steder, må vi ha med denne.

Og her kommer .cpp-filen

```
#include "Arduino.h"
#include "PID.h"

void PID::setConstants ( double _kp, double _ki, double _kd ) {
      kp = _kp;
      //Calculate so the constants maches sample time.
      ki = _ki * sampleTime;
      kd = _kd * sampleTime;
    }

void PID::setKp (double _kp) {
      kp = _kp;

    }
void PID::setKi (double _ki) {
      ki = _ki * sampleTime;
    }
void PID::setKd (double _kd) {
      kd = _kd * sampleTime;
    }

void PID::setLimits( int _minimum, int _maximum ) {
      maximum = (double)_maximum;
      minimum = (double)_minimum;
    }

void PID::setFrequency(uint16_t frequency ) {
      sampleTime = (double) 1 / frequency;
      ki = ki * sampleTime;
      kd = kd * sampleTime;
    }

double PID::compute(double setPoint, double input) {

      //Deres kode her

      return output;
    }
```

Dere skal ikke bekymre dere for mye over syntaksen som blir brukt i koden her. Men det som er viktig her er funksjonen som heter PID::compute(). Dette er hvor dere skal skrive deres egen kode.

Variablene dere skal bruke i koden deres er:

- kp: Proposjonalkonstanten
- ki: Integralkonstanten
- kd: Derivatkonstanten
- maximum: Max verdi som skal returneres
- minimum: Laveste verdi som skal returneres

7 Grafisk brukergrensesnitt

Figur 8: Skjermbilde av brukergrensesnittet

Det grafiske brukergrensesnittet får dere ved å åpne filen Motor_pid_GUI_V2 som ligger i mappen PID-ESP32\Motor_pid_GUI\Motor_pid_GUI_V2\obj\Debug

For å koble til mikrokontrolleren, velger dere riktig COM-port oppe i venstre hjørne og klikker på connect. Forhåpentligvis blir statusen grønn, og dere er oppe og går.

For å få opp grafen klikker dere på start under seksjonen graph.

Når dere skal legge inn verdier, må dere skrive inn verdien i den hvite ruten, trykke på enter, og deretter trykke på update for at verdien skal bli sendt til mikrokontrolleren. Hvis dere lar noen felter være tomme under seksjonen PID-constants vil ikke programmet bli veldig glad, så husk og send inn både P, I og D verdiene første gangen dere skal oppdatere verdiene.

PID sampling frequency endrer samplingsfrekvensen. Denne er satt til å være 100Hz ved oppstart av mikrokontrolleren.

Hvis dere ønsker å bruke potmeteret til å endre settpunkt, huker dere av Use potmeter og trykker update.

Merk: x-aksen er ikke faktisk tid, men antall samples som har blitt gjort.

8 Spørsmål

1. Se på funksjonen void IRAM_ATTR encoderPulse().

   (a) I encoderPulse() finner dere denne linjen med kode:

   ```
   encoderState =  ( 0x0F
           & ( encoderState << 2 ) )
           | ( ( 1 << digitalRead( ENCODER_A ) )
           | digitalRead( ENCODER_B ));
   ```

   I encoderState blir forrige og nåværende tilstand for rotasjonskoderen lagret. Hvordan blir dette gjort? Hva er fordelen og ulempen med å gjøre det på denne måten?

   (b) Hvis encoderState er 00001011 og vi leser inn ENCODER_A=00000000 og ENCODER_B=00000001 i neste itterasjon. Hva blir encoderState da? Hvis utregning.

   (c) Forklar hva hexadesimalverdiene gjør i if og if-else linjene.

2. Se på funksjonen int16_t getValueFromSerial(). Dette er kodesnutten som gjør det mulig og hente kontrollverdier fra PC-en.

   (a) Forklar hvordan koden fungerer(Hint:Ascii)

   (b) Når det blir sendt over en verdi fra PC-en, kan det se ut noe slikt som dette s-50. Denne linjen betyr sett posisjon til -50 grader. Hvis trinnvis hvordan denne strengen av bytes blir lagret i variabelen integerValue.

3. Denne løsningen av PID-regulator bruker kun digitaliserte verdier for å kontrollere motoren. PBM-en er på 10-bit og posisjon kan kun måles i skritt på 0,6 grader.

   (a) Hvordan påvirker dette posisjonsnøyaktigheten vår?

   (b) Hvis vi ikke har mulighet til å endre hardware, kan vi klare å få en bedre nøyaktighet? Hvorfor/hvorfor ikke?

   (c) Hva er integral windup?

   (d) Hvilket verdier i PID.cpp brukes for å motvirke integral windup?

   (e) I PID.cpp blir ki og kd multiplisert med samplingstiden, hvorfor det?

   (f) Hvordan påvirker samplingstiden responsen til systemet?

   (g) Er det alltid fordelaktig å ha høyest mulig samplingfrekvens?

9 Laboppgaver

1. I filen PID.cpp, skal dere skrive PID-koden deres i PID::compute(double setPoint, double input). setPoint er ønsket posisjon i grader og input er faktisk poisjon til motoren. Output er pådraget til motoren, og skal være ett tall mellom _minimum og _maximum. Dette tilsvarer en verdi mellom -1024 og 1024.

   Tips: Ikke skriv all koden på en gang, bygg opp hvert ledd av PID-regulatoren og test mellom hver gang.

2. Det ble nevnt tidligere i teksten her at x-aksen i grafen ikke er den faktiske tiden, noe som er meget ønskelig og ha. Skriv om koden til det grafiske brukergrensesnittet slik at vi får sanntid på grafen. Hvordan dere løser dette er opp til dere.

3. Tune PID-regulatoren. ROBERT!! noe input?